Chapter 7: Declarative Security

Topics in This Chapter

- Understanding the major aspects of Web application security
- Authenticating users with HTML forms
- Using BASIC HTTP authentication
- Defining passwords in Tomcat, JRun, and ServletExec
- Designating protected resources with the `security-constraint` element
- Using `login-config` to specify the authentication method
- Mandating the use of SSL
- Configuring Tomcat to use SSL

Chapter 7

There are two major aspects to securing Web applications:

1. **Preventing unauthorized users from accessing sensitive data.** This process involves *access restriction* (identifying which resources need protection and who should have access to them) and *authentication* (identifying users to determine if they are one of the authorized ones). Simple authentication involves the user entering a username and password in an HTML form or a dialog box; stronger authentication involves the use of X509 certificates sent by the client to the server. This aspect applies to virtually all secure applications. Even intranets at locations with physical access controls usually require some sort of user authentication.
2. **Preventing attackers from stealing network data while it is in transit.** This process involves the use of *Secure Sockets Layer* (SSL) to encrypt the traffic between the browser and the server. This capability is generally reserved for particularly sensitive applications or particularly sensitive pages within a larger application. After all, unless the attackers are on your local subnet, it is exceedingly difficult for them to gain access to your network traffic.

These two security aspects are mostly independent. The approaches to access restriction are the same regardless of whether or not you use SSL. With the exception of client certificates (which apply only to SSL), the approaches to authentication are also identical whether or not you use SSL.

Within the Web application framework, there are two general approaches to this type of security:

1. **Declarative security.** With declarative security, the topic of this chapter, none of the individual servlets or JSP pages need any security-aware code. Instead, both of the major security aspects are handled by the server.

   To prevent unauthorized access, you use the Web application deployment descriptor (*web.xml*) to declare that certain URLs need protection. You also designate the authentication method that the server should use to identify users. At request time, the server automatically prompts users for usernames and passwords when they try to access restricted resources, automatically checks the results against a predefined set of usernames and passwords, and automatically keeps track of which users have previously been authenticated. This process is completely transparent to the servlets and JSP pages.

   To safeguard network data, you use the deployment descriptor to stipulate that certain URLs should only be accessible with SSL. If users try to use a regular HTTP connection to access one of these URLs, the server automatically redirects them to the HTTPS (SSL) equivalent.
2. **Programmatic security.** With programmatic security, the topic of the next chapter, protected servlets and JSP pages at least partially manage their own security.

   To prevent unauthorized access, *each* servlet or JSP page must either authenticate the user or verify that the user has been authenticated previously.

   To safeguard network data, each servlet or JSP page has to check the network protocol used to access it. If users try to use a regular HTTP connection to access one of these URLs, the servlet or JSP page must manually redirect them to the HTTPS (SSL) equivalent.

7.1 Form-Based Authentication

The most common type of declarative security uses regular HTML forms. The developer uses the deployment descriptor to identify the protected resources and to designate a page that has a form to collect usernames and passwords. A user who attempts

to access protected resources is redirected to the page containing the form. When the form is submitted, the server checks the username and password against a list of usernames, passwords and roles. If the login is successful and the user belongs to a role that is permitted access to the page, the user is granted access to the page originally requested. If the login is unsuccessful, the user is sent to a designated error page. Behind the scenes, the system uses some variation of session tracking to remember which users have already been validated.

The whole process is automatic: redirection to the login page, checking of user names and passwords, redirection back to the original resource, and tracking of already authenticated users are all performed by the container (server) in a manner that is completely transparent to the individual resources. However, there is one major caveat: the servlet specification explicitly says that form-based authentication is not guaranteed to work when the server is set to perform session tracking based on URL rewriting instead of cookies (the default session tracking mechanism).

Core Warning

Depending on your server, form-based authentication might fail when you use URL rewriting as the basis of session tracking.

This type of access restriction and authentication is completely independent of the protection of the network traffic. You can stipulate that SSL be used for all, some, or none of your application; but doing so does not change the way you restrict access or authenticate users. Nor does the use of SSL require your individual servlets or JSP pages to participate in the security process; redirection to the URL that uses SSL and encryption/decryption of the network traffic are all performed by the server in a manner that is transparent to the servlets and JSP pages.

Seven basic steps are required to set up your system to use this type of form-based security. I'll summarize the steps here, then give details on each step in the following subsections. All the steps except for the first are standardized and portable across all servers that support version 2.2 or later of the servlet API. Section 7.2 illustrates the concepts with a small application.

1. **Set up usernames, passwords, and roles.** In this step, you designate a list of users and associate each with a password and one or more abstract roles (e.g., normal user or administrator). This is a completely server-specific process. In general, you'll have to read your server's documentation, but I'll summarize the process for Tomcat, JRun, and ServletExec.

2. **Tell the server that you are using form-based authentication. Designate the locations of the login and login-failure page.** This process uses the *web.xml* `login-config` element with an `auth-method` subelement of `FORM` and a `form-login-config` subelement that gives the locations of the two pages.
3. **Create a login page.** This page must have a form with an `ACTION` of `j_security_check`, a `METHOD` of `POST`, a textfield named `j_username`, and a password field named `j_password`.
4. **Create a page to report failed login attempts.** This page can simply say something like "username and password not found" and perhaps give a link back to the login page.
5. **Specify which URLs should be password protected.** For this step, you use the `security-constraint` element of *web.xml*. This element, in turn, uses `web-resource-collection` and `auth-constraint` subelements. The first of these (`web-resource-collection`) designates the URL patterns to which access should be restricted, and the second (`auth-constraint`) specifies the abstract roles that should have access to the resources at the given URLs.
6. **Specify which URLs should be available only with SSL.** If your server supports SSL, you can stipulate that certain resources are available *only* through encrypted HTTPS (SSL) connections. You use the `user-data-constraint` subelement of `security-constraint` for this purpose.
7. **Turn off the invoker servlet.** If your application restricts access to servlets, the access restrictions are placed on the custom URLs that you associate with the servlets. But, most servers have a default servlet URL: *http://host/webAppPrefix/servlet/ServletName*. To prevent users from bypassing the security settings, disable default servlet URLs of this form. To disable these URLs, use the `servlet-mapping` element with a `url-pattern` subelement that designates a pattern of `/servlet/*`.

Details follow.

Setting Up Usernames, Passwords, and Roles

When a user attempts to access a protected resource in an application that is using form-based authentication, the system uses an HTML form to ask for a username and password, verifies that the password matches the user, determines what abstract roles (regular user, administrator, executive, etc.) that user belongs to, and sees whether any of those roles has permission to access the resource. If so, the server

redirects the user to the originally requested page. If not, the server redirects the user to an error page.

The good news regarding this process is that the server (container) does a lot of the work for you. The bad news is that the task of associating users with passwords and logical roles is server specific. So, although you would not have to change the *web.xml* file or any of the actual servlet and JSP code to move a secure Web application from system to system, you would still have to make custom changes on each system to set up the users and passwords.

In general, you will have to read your server's documentation to determine how to assign passwords and role membership to users. However, I'll summarize the process for Tomcat, JRun, and ServletExec.

Setting Passwords with Tomcat

Tomcat permits advanced developers to configure custom username and password management schemes (e.g., by accessing a database, looking in the Unix */etc/passwd* file, checking the Windows NT/2000 User Account settings, or making a Kerberos call). For details, see *http://jakarta.apache.org/tomcat/tomcat-4.0-doc/realm-howto.html*. However, this configuration is a lot of work, so Tomcat also provides a default mechanism. With this mechanism, Tomcat stores usernames, passwords, and roles in *install_dir/conf/tomcat-users.xml*. This file should contain an XML header followed by a `tomcat-users` element containing any number of `user` elements. Each `user` element should have three attributes: `name` (the username), `password` (the plain text password), and `roles` (a comma-separated list of logical role names). Listing 7.1 presents a simple example that defines four users (`valjean`, `bishop`, `javert`, `thenardier`), each of whom belongs to two logical roles.

Listing 7.1 *install_dir/conf/tomcat-users.xml* (Sample)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tomcat-users>
  <user name="valjean" password="forgiven"
        roles="lowStatus,nobleSpirited" />
  <user name="bishop" password="mercy"
        roles="lowStatus,nobleSpirited" />
  <user name="javert" password="strict"
        roles="highStatus,meanSpirited" />
  <user name="thenardier" password="grab"
        roles="lowStatus,meanSpirited" />
</tomcat-users>
```

Note that the default Tomcat strategy of storing unencrypted passwords is a poor one. First, an intruder that gains access to the server's file system can obtain all the passwords. Second, even system administrators who are authorized to access server resources should not be able to obtain user's passwords. In fact, since many users reuse passwords on multiple systems, passwords should *never* be stored in clear text. Instead, they should be encrypted with an algorithm that cannot easily be reversed. Then, when a user supplies a password, it is encrypted and the encrypted version is compared with the stored encrypted password. Nevertheless, the default Tomcat approach makes it easy to set up and test secure Web applications. Just keep in mind that for real applications you'll want to replace the simple file-based password scheme with something more robust (e.g., a database or a system call to Kerberos or the Windows NT/2000 User Account system).

Setting Passwords with JRun

JRun, like Tomcat, permits developers to customize the username and password management scheme. For details, see Chapter 39 (Web Application Authentication) of *http://www.allaire.com/documents/jr31/devapp.pdf*. Also like Tomcat, JRun provides a file-based default mechanism. Unlike Tomcat, however, JRun encrypts the passwords before storing them in the file. This approach makes the default JRun strategy usable even in real-world applications.

With the default mechanism, JRun stores usernames, encrypted passwords, and roles in *install_dir/lib/users.properties*. This file contains entries of three types: `user.`*`username`* entries that associate a password with a user; `group.`*`groupname`* entries that group users together; and `role.`*`rolename`* entries that place users and/or groups into logical roles. Encrypted passwords can be obtained from an existing Unix-based password or *.htaccess* file or by using the `PropertyFileAuthentication` class supplied with JRun. To use this class, temporarily set your `CLASSPATH` (not the server's `CLASSPATH`) to include *install_dir/lib/jrun.jar* and *install_dir/lib/ext/servlet.jar*, change directory to *install_dir/lib*, and add a user at a time with the `-add` flag, as below. For real applications you would probably set up the server to automate this process.

```
java allaire.jrun.security.PropertyFileAuthentication valjean grace
```

After adding the users, edit the file to assign the roles. Listing 7.2 shows an example that sets up the same users, passwords, and roles as in the previous Tomcat example (Listing 7.1).

Listing 7.2 *install_dir/lib/users.properties* (Sample)

```
user.valjean=vaPoR2yIzbfdI
user.bishop=bic5wknlJ8QFE
user.javert=jaLULvqM82wfk
user.thenardier=thvwKJbcM0s7o

role.lowStatus=valjean,thenardier
role.highStatus=bishop,javert
role.nobleSpirited=valjean,bishop
role.meanSpirited=javert,thenardier
```

Setting Passwords with ServletExec

The process of setting up usernames, passwords, and roles is particularly simple with ServletExec. Simply open the administrator home page and select Users within the Web Applications heading (Figure 7–1). From there, you can interactively enter usernames, passwords, and roles (Figure 7–2). Voila!

With the free desktop debugger version, ServletExec stores the usernames and passwords in plain text in *install_dir/ServletExec Data/users.properties*. The passwords are encrypted in the deployment version.

Figure 7–1 ServletExec user editing interface.

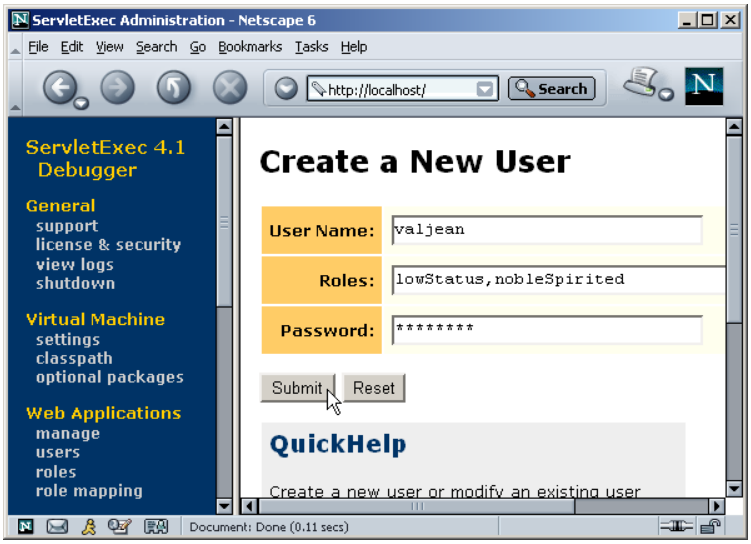

Figure 7–2 Adding a user, password, and role in ServletExec.

2. Telling the Server You Are Using Form-Based Authentication; Designating Locations of Login and Login-Failure Pages

You use the `login-config` element in the deployment descriptor (*web.xml*) to control the authentication method. Recall from Chapters 4 and 5 that this file goes in the *WEB-INF* directory of your Web application. Although a few servers support nonstandard *web.xml* files (e.g., Tomcat has one in *install_dir/conf* that provides defaults for multiple Web applications), those files are entirely server specific. I am addressing only the standard version that goes in the Web application's *WEB-INF* directory.

To use form-based authentication, supply a value of `FORM` for the `auth-method` subelement and use the `form-login-config` subelement to give the locations of the login (`form-login-page`) and login-failure (`form-error-page`) pages. In the next sections I'll explain exactly what these two files should contain. But for now, note that nothing mandates that they use dynamic content. Thus, these pages can consist of either JSP or ordinary HTML.

For example, Listing 7.3 shows part of a *web.xml* file that stipulates that the container use form-based authentication. Unauthenticated users who attempt to access protected resources will be redirected to *http://host/webAppPrefix/login.jsp*. If they log in successfully, they will be returned to whatever resource they first attempted to access. If their login attempt fails, they will be redirected to *http://host/webAppPrefix/login-error.html*.

Listing 7.3 *web.xml* (Excerpt designating form-based authentication)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <!-- ... -->
  <security-constraint>...</security-constraint>
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/login-error.html</form-error-page>
    </form-login-config>
  </login-config>
  <!-- ... -->
</web-app>
```

Creating the Login Page

OK, so the `login-config` element tells the server to use form-based authentication and to redirect unauthenticated users to a designated page. Fine. But what should you put *in* that page? The answer is surprisingly simple: all the login page requires is a form with an `ACTION` of `j_security_check`, a textfield named `j_username`, and a password field named `j_password`. And, since using `GET` defeats the whole point of password fields (protecting the password from prying eyes looking over the user's shoulder), *all* forms that have password fields should use a `METHOD` of `POST`. Note that `j_security_check` is a "magic" name; you don't preface it with a slash even if your login page is in a subdirectory of the main Web application directory. Listing 7.4 gives an example.

Listing 7.4 *login.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>...</TITLE></HEAD>
<BODY>
...
<FORM ACTION="j_security_check" METHOD="POST">
<TABLE>
<TR><TD>User name: <INPUT TYPE="TEXT" NAME="j_username">
<TR><TD>Password: <INPUT TYPE="PASSWORD" NAME="j_password">
<TR><TH><INPUT TYPE="SUBMIT" VALUE="Log In">
</TABLE>
</FORM>
...
</BODY></HTML>
```

OK, that was the page for logging *in*. What about a page for logging *out*? The session should time out eventually, but what if users want to log out immediately without closing the browser? Well, the servlet specification says that invalidating the `HttpSession` should log out users and cause them to be reauthenticated the next time they try to access a protected resource. So, in principle you should be able to create a logout page by making servlet or JSP page that looks up the session and calls `invalidate` on it. In practice, however, not all servers support this process. Fortunately, changing users is simple: you just visit the login page a second time. This is in contrast to BASIC authentication (Section 7.3), where neither logging out nor changing your username is supported without the user quitting and restarting the browser.

2.2 Creating the Page to Report Failed Login Attempts

The main login page must contain a form with a special-purpose `ACTION` (`j_security_check`), a textfield with a special name (`j_username`), and a password field with yet another reserved name (`j_password`). So, what is required to be in the login-failure page? Nothing! This page is arbitrary; it can contain a link to an unrestricted section of the Web application, a link to the login page, or a simple "login failed" message.

3. Specifying URLs That Should Be Password Protected

The `login-config` element tells the server which authentication method to use. Good, but how do you designate the specific URLs to which access should be

restricted? Designating restricted URLs and describing the protection they should have is the purpose of the `security-constraint` element.

The `security-constraint` element should come immediately before `login-config` in *web.xml* and contains four possible subelements: `display-name` (an optional element giving a name for IDEs to use), `web-resource-collection` (a required element that specifies the URLs that should be protected), `auth-constraint` (an optional element that designates the abstract roles that should have access to the URLs), and `user-data-constraint` (an optional element that specifies whether SSL is required). Note that multiple `web-resource-collection` entries are permitted within `security-constraint`.

For a quick example of the use of `security-constraint`, Listing 7.5 instructs the server to require passwords for all URLs of the form *http://host/webAppPrefix/sensitive/blah*. Users who supply passwords and belong to the `administrator` or `executive` logical roles should be granted access; all others should be denied access. The rest of this subsection provides details on the `web-resource-collection`, `auth-constraint`, and `display-name` elements. The role of `user-data-constraint` is explained in the next subsection (Specifying URLs That Should Be Available Only with SSL).

Listing 7.5 *web.xml* (Excerpt specifying protected URLs)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <!-- ... -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Sensitive</web-resource-name>
      <url-pattern>/sensitive/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>administrator</role-name>
      <role-name>executive</role-name>
    </auth-constraint>
  </security-constraint>
  <login-config>...</login-config>
  <!-- ... -->
</web-app>
```

display-name

This rarely used optional subelement of `security-constraint` gives a name to the security constraint entry. This name might be used by an IDE or other graphical tool.

web-resource-collection

This subelement of `security-constraint` identifies the resources that should be protected. Each `security-constraint` element must contain one or more `web-resource-collection` entries; all other `security-constraint` subelements are optional. The `web-resource-collection` element consists of a `web-resource-name` element that gives an arbitrary identifying name, a `url-pattern` element that identifies the URLs that should be protected, an optional `http-method` element that designates the HTTP commands to which the protection applies (`GET`, `POST`, etc.; the default is all methods), and an optional `description` element providing documentation. For example, the following `web-resource-collection` entries (within a `security-constraint` element) specify that password protection should be applied to all documents in the *proprietary* directory (and subdirectories thereof) and to the *delete-account.jsp* page in the *admin* directory.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Proprietary</web-resource-name>
    <url-pattern>/proprietary/*</url-pattern>
  </web-resource-collection>
  <web-resource-collection>
    <web-resource-name>Account Deletion</web-resource-name>
    <url-pattern>/admin/delete-account.jsp</url-pattern>
  </web-resource-collection>
  <!-- ... -->
</security-constraint>
```

It is important to note that the `url-pattern` applies only to clients that access the resources directly. In particular, it does *not* apply to pages that are accessed through the MVC architecture with a `RequestDispatcher` (Section 3.8) or by the similar means of `jsp:forward` or `jsp:include` (Section 3.5). This asymmetry is good if used properly. For example, with the MVC architecture a servlet looks up data, places it in beans, and forwards the request to a JSP page that extracts the data from the beans and displays it. You want to ensure that the JSP page is never accessed directly but instead is accessed only through the servlet that sets up the beans the page will use. The `url-pattern` and `auth-constraint` (see next subsection) elements can provide this guarantee by declaring that *no* user is permitted direct access to the JSP page. But, this asymmetric behavior can catch developers off guard and allow them to accidentally provide unrestricted access to resources that should be protected.

Core Warning

These protections apply only to direct client access. The security model does not apply to pages accessed by means of a `RequestDispatcher`*,* `jsp:forward`*, or* `jsp:include`*.*

auth-constraint

Whereas the `web-resource-collection` element designates the URLs that should be protected, the `auth-constraint` element designates the users that should have access to protected resources. It should contain one or more `role-name` elements identifying the class of users that have access and, optionally, a `description` element describing the role. For instance, the following part of the `security-constraint` element in *web.xml* states that only users who are designated as either Administrators or Big Kahunas (or both) should have access to the designated resource.

```
<security-constraint>
  <web-resource-collection>...</web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
    <role-name>kahuna</role-name>
  </auth-constraint>
</security-constraint>
```

If you want all authenticated users to have access to a resource, use `*` as the `role-name`. Technically, the `auth-constraint` element is optional. Omitting it means that *no* roles have access. Although at first glance it appears pointless to deny access to all users, remember that these security restrictions apply only to direct client access. So, for example, suppose you had a JSP snippet that is intended to be inserted into another file with `jsp:include` (Section 3.5). Or, suppose you have a JSP page that is the forwarding destination of a servlet that is using a `RequestDispatcher` as part of the MVC architecture (Section 3.8). In both cases, users should be prohibited from directly accessing the JSP page. A `security-constraint` element with no `auth-constraint` would enforce this restriction nicely.

3.1 Specifying URLs That Should Be Available Only with SSL

Suppose your servlet or JSP page collects credit card numbers. User authentication keeps out unauthorized users but does nothing to protect the network traffic. So, for instance, an attacker that runs a packet sniffer on the end user's local area network

could see that user's credit card number. This scenario is exactly what SSL protects against—it encrypts the traffic between the browser and the server.

Use of SSL does not change the basic way that form-based authentication works. Regardless of whether you are using SSL, you use the `login-config` element to indicate that you are using form-based authentication and to identify the login and login-failure pages. With or without SSL, you designate the protected resources with the `url-pattern` subelement of `web-resource-collection`. None of your servlets or JSP pages need to be modified or moved to different locations when you enable or disable SSL. That's the beauty of declarative security.

The `user-data-constraint` subelement of `security-constraint` can mandate that certain resources be accessed only with SSL. So, for example, attempts to access *https://host/webAppPrefix/specialURL* are handled normally, whereas attempts to access *http://host/webAppPrefix/specialURL* are redirected to the *https* URL. This behavior does not mean that you cannot supply an explicit *https* URL for a hypertext link or the `ACTION` of a form; it just means that you aren't *required* to. You can stick with the simpler and more easily maintained relative URLs and still be assured that certain URLs will only be accessed with SSL.

The `user-data-constraint` element, if used, must contain a `transport-guarantee` subelement (with legal values `NONE`, `INTEGRAL`, or `CONFIDENTIAL`) and can optionally contain a `description` element. A value of `NONE` for `transport-guarantee` puts no restrictions on the communication protocol used. Since `NONE` is the default, there is little point in using `user-data-constraint` or `transport-guarantee` if you specify `NONE`. A value of `INTEGRAL` means that the communication must be of a variety that prevents data from being changed in transit without detection. A value of `CONFIDENTIAL` means that the data must be transmitted in a way that prevents anyone who intercepts it from reading it. Although in principle (and perhaps in future HTTP versions) there may be a distinction between `INTEGRAL` and `CONFIDENTIAL`, in current practice they both simply mandate the use of SSL.

For example, the following instructs the server to permit only *https* connections to the associated resource:

```
<security-constraint>
  <!-- ... -->
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

In addition to simply requiring SSL, the servlet API provides a way to stipulate that users must authenticate themselves with client certificates. You supply a value of `CLIENT-CERT` for the `auth-method` subelement of `login-config` (see "Specifying URLs That Should Be Password Protected" earlier in this section). However, only servers that have full J2EE support are required to support this capability.

Now, although the method of prohibiting non-SSL access is standardized, servers that are compliant with the servlet 2.3 and JSP 1.2 specifications are *not* required to support SSL. So, Web applications that use a `transport-guarantee` of `CONFIDENTIAL` (or, equivalently, `INTEGRAL`) are not necessarily portable. For example, JRun and ServletExec are usually used as plugins in Web servers like iPlanet/Netscape or IIS. In this scenario, the network traffic between the client and the Web server is encrypted with SSL, but the local traffic from the Web server to the servlet/JSP container is not encrypted. Consequently, a `CONFIDENTIAL transport-guarantee` will fail. Tomcat, however, can be set up to use SSL directly. Details on this process are given in Section 7.5. Some server plugins maintain SSL even on the local connection between the main Web server and the servlet/JSP engine; for example, the BEA WebLogic plugin for IIS, Apache, and Netscape Enterprise Server does so. Furthermore, integrated application servers like the standalone version of WebLogic have no "separate" servlet and JSP engine, so SSL works exactly as described here. Nevertheless, it is important to realize that these features, although useful, are not mandated by the servlet and JSP specifications.

Core Warning

Web applications that rely on SSL are not necessarily portable.

4. Turning Off the Invoker Servlet

When you restrict access to certain resources, you do so by specifying the URL patterns to which the restrictions apply. This pattern, in turn, matches a pattern that you set with the `servlet-mapping` *web.xml* element (see Section 5.3, "Assigning Names and Custom URLs"). However, most servers use an "invoker servlet" that provides a default URL for servlets: *http://host/webAppPrefix/servlet/ServletName*. You need to make sure that users don't access protected servlets with this URL, thus bypassing the access restrictions that were set by the `url-pattern` subelement of `web-resource-collection`.

For example, suppose that you use `security-constraint`, `web-resource-collection`, and `url-pattern` to say that the URL */admin/DeclareChapter11* should be protected. You also use the `auth-constraint` and `role-name` elements to say that only users in the `director` role can access this URL. Next, you use the `servlet` and `servlet-mapping` elements to say that the servlet *BankruptcyServlet.class* in the `disaster` package should correspond to */admin/DeclareChapter11*. Now, the security restrictions are in force when clients use the URL *http://host/webAppPrefix/admin/DeclareChapter11*. No restrictions apply to *http://host/webAppPrefix/servlet/disaster.BankruptcyServlet*. Oops.

Section 5.4 (Disabling the Invoker Servlet) discusses server-specific approaches to turning off the invoker. The most portable approach, however, is to simply remap the */servlet* pattern in your Web application so that all requests that include the pattern are sent to the same servlet. To remap the pattern, you first create a simple servlet that prints an error message or redirects users to the top-level page. Then, you use the `servlet` and `servlet-mapping` elements (Section 5.3) to send requests that include the */servlet* pattern to that servlet. Listing 7.6 gives a brief example.

Listing 7.6 *web.xml* (Excerpt redirecting requests from default servlet URLs to an error-message servlet)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>Error</servlet-name>
    <servlet-class>somePackage.ErrorServlet</servlet-class>
  </servlet>
  <!-- ... -->
  <servlet-mapping>
    <servlet-name>Error</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
  <!-- ... -->
</web-app>
```

7.2 Example: Form-Based Authentication

In this section I'll work through a small Web site for a fictional company called hot-dot-com.com. I'll start by showing the home page, then list the *web.xml* file, summarize the various protection mechanisms, show the password file, present the login and login-failure pages, and give the code for each of the protected resources.

The Home Page

Listing 7.7 shows the top-level home page for the Web application. The application is registered with a URL prefix of */hotdotcom* so the home page can be accessed with the URL *http://host/hotdotcom/index.jsp* as shown in Figure 7–3. If you've forgotten how to assign URL prefixes to Web applications, review Section 4.1 (Registering Web Applications).

Now, the main home page has no security protections and consequently does not absolutely require an entry in *web.xml*. However, many users expect URLs that list a directory but no file to invoke the default file from that directory. So, I put a `welcome-file-list` entry in *web.xml* (see Listing 7.8 in the next section) to ensure that *http://host/hotdotcom/* would invoke *index.jsp*.

Listing 7.7 *index.jsp* (Top-level home page)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>hot-dot-com.com!</TITLE>
<LINK REL=STYLESHEET
      HREF="company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">hot-dot-com.com!</TABLE>
<P>
<H3>Welcome to the ultimate dot-com company!</H3>
Please select one of the following:
<UL>
  <LI><A HREF="investing/">Investing</A>.
      Guaranteed growth for your hard-earned dollars!
  <LI><A HREF="business/">Business Model</A>.
      New economy strategy!
  <LI><A HREF="history/">History</A>.
      Fascinating company history.
</UL>
</BODY>
</HTML>
```

Figure 7–3 Home page for hot-dot-com.com.

The Deployment Descriptor

Listing 7.8 shows the complete deployment descriptor used with the `hotdotcom` Web application. Recall that the order of the subelements within the `web-app` element of *web.xml* is not arbitrary—you must use the standard ordering. For details, see Section 5.2 (The Order of Elements within the Deployment Descriptor).

The `hotdotcom` deployment descriptor specifies several things:

- URLs that give a directory but no filename result in the server first trying to use *index.jsp* and next trying *index.html*. If neither file is available, the result is server specific (e.g., a directory listing).
- URLs that use the default servlet mapping (i.e., *http://host/hotdotcom/servlet/ServletName*) are redirected to the main home page.
- Requests to *http://host/hotdotcom/ssl/buy-stock.jsp* are redirected to *https://host/hotdotcom/ssl/buy-stock.jsp*. Requests directly to *https://host/hotdotcom/ssl/buy-stock.jsp* require no redirection. Similarly, requests to *http://host/hotdotcom/ssl/FinalizePurchase* are redirected to *https://host/hotdotcom/ssl/FinalizePurchase*. See Section 7.5 for information on setting up Tomcat to use SSL.
- URLs in the *investing* directory can be accessed only by users in the `registered-user` or `administrator` roles.
- The *delete-account.jsp* page in the *admin* directory can be accessed only by users in the `administrator` role.

- Requests for restricted resources by unauthenticated users are redirected to the *login.jsp* page in the *admin* directory. Users who are authenticated successfully get sent to the page they tried to access originally. Users who fail authentication are sent to the *login-error.jsp* page in the *admin* directory.

Listing 7.8 *WEB-INF/web.xml* (Complete version for *hot-dot-com.com*)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <!-- Give name to FinalizePurchaseServlet. This servlet
       will later be mapped to the URL /ssl/FinalizePurchase
       (by means of servlet-mapping and url-pattern).
       Then, that URL will be designated as one requiring
       SSL (by means of security-constraint and
       transport-guarantee). -->
  <servlet>
    <servlet-name>
      FinalizePurchaseServlet
    </servlet-name>
    <servlet-class>
      hotdotcom.FinalizePurchaseServlet
    </servlet-class>
  </servlet>

  <!-- A servlet that redirects users to the home page. -->
  <servlet>
    <servlet-name>Redirector</servlet-name>
    <servlet-class>hotdotcom.RedirectorServlet</servlet-class>
  </servlet>

  <!-- Associate previously named servlet with custom URL. -->
  <servlet-mapping>
    <servlet-name>
      FinalizePurchaseServlet
    </servlet-name>
    <url-pattern>
      /ssl/FinalizePurchase
    </url-pattern>
  </servlet-mapping>
```

Listing 7.8 *WEB-INF/web.xml* (Complete version for *hot-dot-com.com*) *(continued)*

```
<!-- Turn off invoker. Send requests to index.jsp. -->
<servlet-mapping>
  <servlet-name>Redirector</servlet-name>
  <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>

<!-- If URL gives a directory but no filename, try index.jsp
     first and index.html second. If neither is found,
     the result is server-specific (e.g., a directory
     listing). -->
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>

<!-- Protect everything within the "investing" directory. -->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Investing</web-resource-name>
    <url-pattern>/investing/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>registered-user</role-name>
    <role-name>administrator</role-name>
  </auth-constraint>
</security-constraint>

<!-- URLs of the form http://host/webAppPrefix/ssl/blah
     require SSL and are thus redirected to
     https://host/webAppPrefix/ssl/blah. -->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Purchase</web-resource-name>
    <url-pattern>/ssl/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>registered-user</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Listing 7.8 *WEB-INF/web.xml* (Complete version for *hot-dot-com.com*) *(continued)*

```
  <!-- Only users in the administrator role can access
       the delete-account.jsp page within the admin
       directory. -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Account Deletion</web-resource-name>
      <url-pattern>/admin/delete-account.jsp</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>administrator</role-name>
    </auth-constraint>
  </security-constraint>

  <!-- Tell the server to use form-based authentication. -->
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/admin/login.jsp</form-login-page>
      <form-error-page>/admin/login-error.jsp</form-error-page>
    </form-login-config>
  </login-config>
</web-app>
```

The Password File

With form-based authentication, the server (container) performs a lot of the work for you. That's good. However, shifting so much work to the server means that there is a server-specific component: the assignment of passwords and roles to individual users (see Section 7.1).

Listing 7.9 shows the password file used by Tomcat for this Web application. It defines four users: `john` (in the `registered-user` role), `jane` (also in the `registered-user` role), `juan` (in the `administrator` role), and `juana` (in the `registered-user` and `administrator` roles).

Listing 7.9 *install_dir/conf/tomcat-users.xml* (First four users)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tomcat-users>
  <user name="john" password="nhoj"
        roles="registered-user" />
  <user name="jane" password="enaj"
        roles="registered-user" />
  <user name="juan" password="nauj"
        roles="administrator" />
  <user name="juana" password="anauj"
        roles="administrator,registered-user" />
</tomcat-users>
```

The Login and Login-Failure Pages

This Web application uses form-based authentication. Attempts by not-yet-authenticated users to access any password-protected resource will be sent to the *login.jsp* page in the *admin* directory. This page, shown in Listing 7.10, collects the username in a field named `j_username` and the password in a field named `j_password`. The results are sent by `POST` to a resource called `j_security_check`. Successful login attempts are redirected to the page that was originally requested. Failed attempts are redirected to the *login-error.jsp* page in the *admin* directory (Listing 7.11).

Listing 7.10 *admin/login.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Log In</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Log In</TABLE>
<P>
<H3>Sorry, you must log in before accessing this resource.</H3>
<FORM ACTION="j_security_check" METHOD="POST">
```

Listing 7.10 *admin/login.jsp (continued)*

```
<TABLE>
<TR><TD>User name: <INPUT TYPE="TEXT" NAME="j_username">
<TR><TD>Password: <INPUT TYPE="PASSWORD" NAME="j_password">
<TR><TH><INPUT TYPE="SUBMIT" VALUE="Log In">
</TABLE>
</FORM>

</BODY>
</HTML>
```

Listing 7.11 *admin/login-error.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Begone!</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Begone!</TABLE>

<H3>Begone, ye unauthorized peon.</H3>

</BODY>
</HTML>
```

The investing Directory

The *web.xml* file for the `hotdotcom` Web application (Listing 7.8) specifies that all URLs that begin with *http://host/hotdotcom/investing/* should be password protected, accessible only to users in the `registered-user` role. So, the first attempt by any user to access the home page of the *investing* directory (Listing 7.12) results in the login form shown earlier in Listing 7.10. Figure 7–4 shows the initial result, Figure 7–5 shows the result of an unsuccessful login attempt, and Figure 7–6 shows the investing home page—the result of a successful login.

Once authenticated, a user can browse other pages and return to a protected page without reauthentication. The system uses some variation of session tracking to remember which users have previously been authenticated.

Listing 7.12 *investing/index.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Investing</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Investing</TABLE>
<H3><I>hot-dot-com.com</I> welcomes the discriminating investor!
</H3>
Please choose one of the following:
<UL>
  <LI><A HREF="../ssl/buy-stock.jsp">Buy stock</A>.
      Astronomic growth rates!
  <LI><A HREF="account-status.jsp">Check account status</A>.
      See how much you've already earned!
</UL>
</BODY>
</HTML>
```

Figure 7–4 Users who are not yet authenticated get redirected to the login page when they attempt to access the investing page.

Figure 7–5 Failed login attempts result in the *login-error.jsp* page. Internet Explorer users have to turn off "friendly" HTTP error messages (under Tools, Internet Options, Advanced) to see the real error page instead of a default error page.

Figure 7–6 Successful login attempts result in redirection back to the originally requested page.

The ssl Directory

The stock purchase page (Listings 7.13 and 7.14) submits data to the purchase finalization servlet (Listing 7.15) which, in turn, dispatches to the confirmation page (Listing 7.16).

Note that the purchase finalization servlet is not really in the *ssl* directory; it is in *WEB-INF/classes/hotdotcom*. However, the deployment descriptor (Listing 7.8) uses `servlet-mapping` to assign a URL that makes the servlet appear (to the client) to be in the *ssl* directory. This mapping serves two purposes.

First, it lets the HTML form of Listing 7.13 use a simple relative URL to refer to the servlet. This is convenient because absolute URLs require modification every time your hostname or URL prefix changes. However, if you use this approach, it is important that *both* the original form and the servlet it talks to are accessed with SSL. If the original form used a relative URL for the `ACTION` and was accessed with a normal HTTP connection, the browser would first submit the data by HTTP and then get redirected to HTTPS. Too late: an attacker with access to the network traffic could have obtained the data from the initial HTTP request. On the other hand, if the `ACTION` of a form is an absolute URL that uses *https*, it is not necessary for the original form to be accessed with SSL.

Second, using `servlet-mapping` in this way guarantees that SSL will be used to access the servlet, even if the user tries to bypass the HTML form and access the servlet URL directly. This guarantee is in effect because the `transport-guarantee` element (with a value of `CONFIDENTIAL`) applies to the pattern `/ssl/*`. Figures 7–7 through 7–9 show the results.

Listing 7.13 *ssl/buy-stock.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Purchase</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Purchase</TABLE>
<P>
<H3><I>hot-dot-com.com</I> congratulates you on a wise
investment!</H3>
<jsp:useBean id="stock" class="hotdotcom.StockInfo" />
```

Listing 7.13 *ssl/buy-stock.jsp (continued)*

```
<UL>
  <LI>Current stock value:
      <jsp:getProperty name="stock" property="currentValue" />
  <LI>Predicted value in one year:
      <jsp:getProperty name="stock" property="futureValue" />
</UL>
<FORM ACTION="FinalizePurchase" METHOD="POST">
  <DL>
    <DT>Number of shares:
    <DD><INPUT TYPE="RADIO" NAME="numShares" VALUE="1000">
        1000
    <DD><INPUT TYPE="RADIO" NAME="numShares" VALUE="10000">
        10000
    <DD><INPUT TYPE="RADIO" NAME="numShares" VALUE="100000"
               CHECKED>
        100000
  </DL>
  Full name: <INPUT TYPE="TEXT" NAME="fullName"><BR>
  Credit card number: <INPUT TYPE="TEXT" NAME="cardNum"><P>
  <CENTER><INPUT TYPE="SUBMIT" VALUE="Confirm Purchase"></CENTER>
</FORM>
</BODY>
</HTML>
```

Listing 7.14 *WEB-INF/classes/hotdotcom/StockInfo.java* (Bean used by *buy-stock.jsp*)

```
package hotdotcom;

public class StockInfo {
  public String getCurrentValue() {
    return("$2.00");
  }

  public String getFutureValue() {
    return("$200.00");
  }
}
```

Listing 7.15 *WEB-INF/classes/hotdotcom/FinalizePurchaseServlet.java*

```
package hotdotcom;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet that reads credit card information,
 *  performs a stock purchase, and displays confirmation page.
 */

public class FinalizePurchaseServlet extends HttpServlet {

  /** Use doPost for non-SSL access to prevent
   *  credit card number from showing up in URL.
   */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    String fullName = request.getParameter("fullName");
    String cardNum = request.getParameter("cardNum");
    confirmPurchase(fullName, cardNum);
    String destination = "/investing/sucker.jsp";
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher(destination);
    dispatcher.forward(request, response);
  }

  /** doGet calls doPost. Servlets that are
   *  redirected to through SSL must have doGet.
   */

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    doPost(request, response);
  }

  private void confirmPurchase(String fullName,
                               String cardNum) {
    // Details removed to protect the guilty.
  }
}
```

Listing 7.16 *investing/sucker.jsp* (Dispatched to from *FinalizePurchaseServlet.java*)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Thanks!</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Thanks!</TABLE>
<H3><I>hot-dot-com.com</I> thanks you for your purchase.</H3>
You'll be thanking yourself soon!
</BODY>
</HTML>
```

Figure 7–7 Warning when user first accesses `FinalizePurchaseServlet` when Tomcat is using a self-signed certificate. Self-signed certificates result in warnings and are for test purposes only. See Section 7.5 for details on creating them for use with Tomcat and for information on suppressing warnings for future requests.

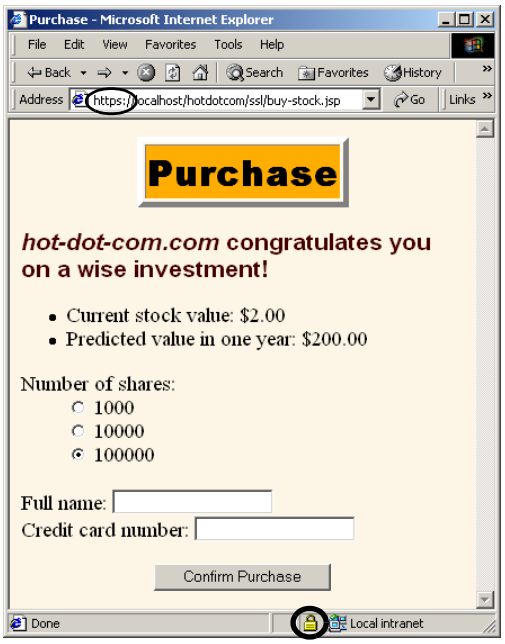

Figure 7–8 The stock purchase page must be accessed with SSL. Since the form's `ACTION` uses a simple relative URL, the initial form submission uses the same protocol as the request for the form itself. If you were concerned about overloading your SSL server (HTTPS connections are much slower than HTTP connections), you could access the form with a non-SSL connection and then supply an absolute URL specifying *https* for the form's `ACTION`. This approach, although slightly more efficient, is significantly harder to maintain.

Figure 7–9 To protect the credit card number in transit, you must use SSL to access the `FinalizePurchase` servlet. Although `FinalizePurchaseServlet` dispatches to *sucker.jsp*, no *web.xml* entry is needed for that JSP page. Access restrictions apply to the client's URL, not to the behind-the-scenes file locations.

Listing 7.17 *investing/account-status.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Account Status</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Account Status</TABLE>
<P>
<H3>Your stock is basically worthless now.</H3>
But, hey, that makes this a buying opportunity.
Why don't you <A HREF="../ssl/buy-stock.jsp">buy
some more</A>?
</BODY>
</HTML>
```

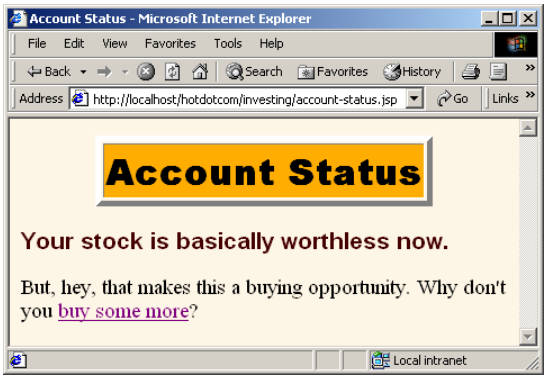

Figure 7–10 Selecting the Account Status link on the investing home page does not result in reauthentication, even if the user has accessed other pages since being authenticated. The system uses a variation of session tracking to remember which users have already been authenticated.

The admin Directory

URLs in the *admin* directory are not uniformly protected as are URLs in the *investing* directory. I already discussed the login and login-failure pages (Listings 7.10 and 7.11, Figures 7–4 and 7–5). This just leaves the Delete Account page (Listing 7.18). This page has been designated as accessible only to users in the `administrator` role. So, when users that are only in the `registered-user` role attempt to access the page, they are denied permission (see Figure 7–11). Note that the permission-denied page of Figure 7–11 is generated automatically by the server and applies to authenticated users whose roles do not match any of the required ones—it is not the same as the login error page that applies to users who cannot be authenticated.

A user in the `administrator` role can access the page without difficulty (Figure 7–12).

Listing 7.18 *admin/delete-account.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Delete Account</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Delete Account</TABLE>
<P>

<FORM ACTION="confirm-deletion.jsp">
  Username: <INPUT TYPE="TEXT" NAME="userName"><BR>
  <CENTER><INPUT TYPE="SUBMIT" VALUE="Confirm Deletion"></CENTER>
</FORM>

</BODY>
</HTML>
```

Figure 7–11 When John and Jane attempt to access the Delete Account page, they are denied (even though they are authenticated). That's because they belong to the `registered-user` role and the *web.xml* file stipulates that only users in the `administrator` role should be able to access this page.

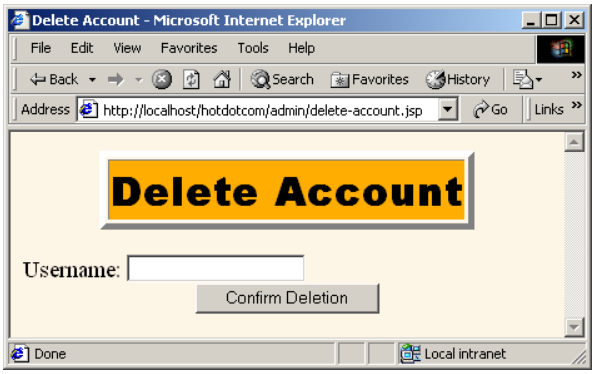

Figure 7–12 Once authenticated, Juan or Juana (in the `administrator` role) can access the Delete Account page.

The Redirector Servlet

Web applications that have protected servlets should always disable the invoker servlet so that users cannot bypass security by using *http://host/webAppPrefix/servlet/ServletName* when the access restrictions are assigned to a custom servlet URL. In the `hotdotcom` application, I used the `servlet` and `servlet-mapping` elements

to register the `RedirectorServlet` with requests to *http://host/hotdotcom/servlet/anything*. This servlet, shown in Listing 7.19, simply redirects all such requests to the application's home page.

Listing 7.19 *WEB-INF/classes/hotdotcom/RedirectorServlet.java*

```
package hotdotcom;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet that simply redirects users to the
 *  Web application home page. Registered with the
 *  default servlet URL to prevent access to servlets
 *  through URLs that have no security settings.
 */

public class RedirectorServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.sendRedirect(request.getContextPath());
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Unprotected Pages

The fact that *some* pages in a Web application have access restrictions does not imply that *all* pages in the application need such restrictions. Resources that have no access restrictions need no special handling regarding security. There are two points to keep in mind, however.

First, if you use default pages such as *index.jsp* or *index.html*, you should have an explicit `welcome-file-list` entry in *web.xml*. Without a `welcome-file-list` entry, servers are not required to use those files as the default file when a user supplies a URL that gives only a directory. See Section 5.7 (Specifying Welcome Pages) for details on the `welcome-file-list` element.

Second, you should use relative URLs to refer to images or style sheets so that your pages don't need modification if the Web application's URL prefix changes. For more information, see Section 4.5 (Handling Relative URLs in Web Applications).

Listings 7.20 and 7.21 (Figures 7–13 and 7–14) give two examples.

Listing 7.20 *business/index.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Business Model</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Business Model</TABLE>
<P>
<H3>Who needs a business model?</H3>
Hey, this is the new economy. We don't need a real business
model, do we?
<P>
OK, ok, if you insist:
<OL>
  <LI>Start a dot-com.
  <LI>Have an IPO.
  <LI>Get a bunch of suckers to work for peanuts
      plus stock options.
  <LI>Retire.
</OL>
Isn't that what many other dot-coms did?
</BODY>
</HTML>
```

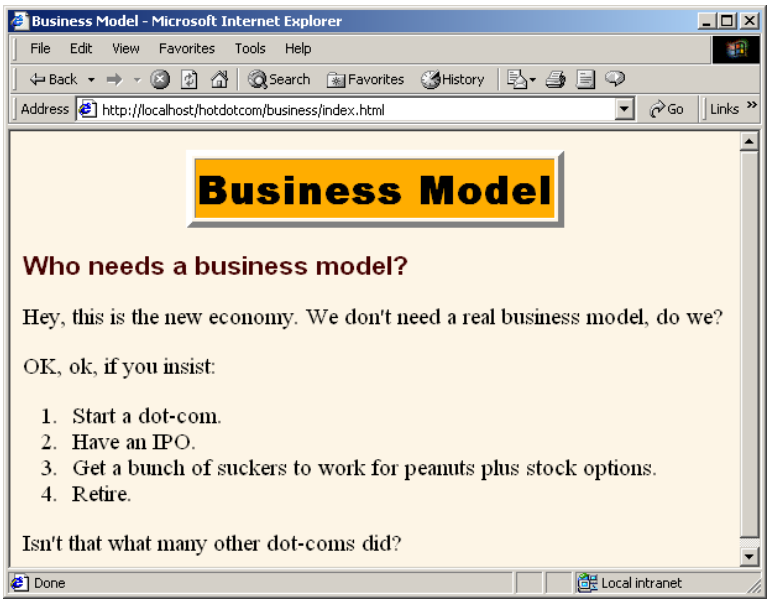

Figure 7–13 The `hotdotcom` business model.

Listing 7.21 *history/index.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>History</TITLE>
<LINK REL=STYLESHEET
      HREF="../company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">History</TABLE>
<P>
<H3>None yet...</H3>
</BODY>
</HTML>
```

Figure 7–14 The distinguished `hotdotcom` heritage.

7.3 BASIC Authentication

The most common type of container-managed security is built on form-based authentication, discussed in Section 7.1. There, the server automatically redirects unauthenticated users to an HTML form, checks their username and password, determines which logical roles they are in, and sees whether any of those roles is permitted to access the resource in question. Then, it uses a variation of session tracking to remember the users that have already been authenticated.

This approach has the advantage that the login form can have the same look and feel as the rest of the Web site. However, it has a few disadvantages. For example, if the client's browser does not support cookies, session tracking would have to be performed with URL rewriting. Or, the server might be configured to always use URL rewriting. The servlet specification explicitly states that form-based authentication is not guaranteed to work in such a case.

So, another approach is to use the standard HTTP BASIC security. With BASIC security, the browser uses a dialog box instead of an HTML form to collect the username and password. Then, the `Authorization` request header is used to remember which users have already been authenticated. As with form-based security, you must use SSL if you are concerned with protecting the network traffic. However, doing so neither changes the way BASIC authentication is set up nor necessitates changes in the individual servlets or JSP pages.

There is also DIGEST security and security based on client certificates. However, few browsers or servers support DIGEST, and only fully J2EE-compliant servers are required to support client certificates. For more information on client certificates, see Section 8.5 (Using Programmatic Security with SSL).

Compared to form-based authentication, the two main disadvantages of BASIC authentication are that the input dialog looks glaringly different than the rest of your application and that it is very difficult to log in as a different user once you are authenticated. In fact, once authenticated, you have to quit the browser and restart if you want to log in as a different user! Now, in principle it is possible to write a "relogin" servlet that sends a 401 (`Unauthorized`) status code and a `WWW-Authenticate` header containing the appropriate realm. But, that is hardly "declarative" security!

Use of BASIC security involves five steps, as shown below. Each of the steps except for the second is identical to the corresponding step used in form-based authentication.

1. **Set up usernames, passwords, and roles.** In this step, you designate a list of users and associate each with a password and one or more abstract roles (e.g., normal user, administrator, etc.). This is a completely server-specific process.
2. **Tell the server that you are using BASIC authentication. Designate the realm name.** This process uses the *web.xml* `login-config` element with an `auth-method` subelement of `BASIC` and a `realm-name` subelement that specifies the realm (which is generally used as part of the title of the dialog box that the browser opens).
3. **Specify which URLs should be password protected.** For this step, you use the `security-constraint` element of *web.xml*. This element, in turn, uses `web-resource-collection` and `auth-constraint` subelements. The first of these designates the URL patterns to which access should be restricted, and the second specifies the abstract roles that should have access to the resources at the given URLs.
4. **Specify which URLs should be available only with SSL.** If your server supports SSL, you can stipulate that certain resources are available *only* through encrypted *https* (SSL) connections. You use the `user-data-constraint` subelement of `security-constraint` for this purpose.
5. **Turn off the invoker servlet.** If your application restricts access to servlets, the access restrictions are placed only on the custom URL that you associate with the servlet. To prevent users from bypassing the security settings, disable default servlet URLs of the form *http://host/webAppPrefix/servlet/ServletName*. To disable these URLs, use the `servlet-mapping` element with a `url-pattern` subelement that designates a pattern of `/servlet/*`.

Details on these steps are given in the following sections.

Setting Up Usernames, Passwords, and Roles

This step is exactly the same when BASIC authentication is used as when form-based authentication is used. See Section 7.1 for details. For a quick summary, recall that this process is completely server specific. Tomcat uses *install_dir/conf/tomcat-users.xml* to store this information, JRun uses *install_dir/lib/users.properties*, and ServletExec has an interactive user interface to enable you to specify the information.

Telling the Server You Are Using BASIC Authentication; Designating Realm

You use the `login-config` element in the deployment descriptor to control the authentication method. To use BASIC authentication, supply a value of `BASIC` for the `auth-method` subelement and use the `realm-name` subelement to designate the realm that will be used by the browser in the popup dialog box and in the `Authorization` request header. Listing 7.22 gives an example.

Listing 7.22 *web.xml* (Excerpt designating BASIC authentication)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <!-- ... -->
  <security-constraint>...</security-constraint>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Some Name</realm-name>
  </login-config>
  <!-- ... -->
</web-app>
```

Specifying URLs That Should Be Password Protected

You designate password-protected resources in the same manner with BASIC authentication as you do with form-based authentication. See Section 7.1 for details. For a

quick summary, you use the `security-constraint` element to specify restricted URLs and the roles that should have access to them. The `security-constraint` element should come immediately before `login-config` in *web.xml* and contains four possible subelements: `display-name` (an optional element giving a name for IDEs to use), `web-resource-collection` (a required element that specifies the URLs that should be protected), `auth-constraint` (an optional element that designates the abstract roles that should have access to the URLs), and `user-data-constraint` (an optional element that specifies whether SSL is required). Multiple `web-resource-collection` entries are permitted within `security-constraint`.

Specifying URLs That Should Be Available Only with SSL

You designate SSL-only resources in the same manner with BASIC authentication as you do with form-based authentication. See Section 7.1 for details. To summarize: use the `user-data-constraint` subelement of `security-constraint` with a `transport-guarantee` subelement specifying `INTEGRAL` or `CONFIDENTIAL`.

In addition to simply requiring SSL, the servlet API provides a way for stipulating that users must authenticate themselves with client certificates. You supply a value of `CLIENT-CERT` for the `auth-method` subelement of `login-config` (see "Specifying URLs That Should Be Password Protected" in Section 7.1). However, only application servers that have full J2EE support are required to support this capability.

7.4 Example: BASIC Authentication

In Section 7.2, I showed the external Web site for a fictional company named hot-dot-com.com. In this section, I'll show their intranet. Since applications that use form-based authentication vary only slightly from those that use BASIC authentication, I'll just concentrate on the differences here. I'll start by showing the home page, then list the *web.xml* file, summarize the various protection mechanisms, show the password file, and give the code for each of the protected resources.

The Home Page

Listing 7.23 shows the top-level home page for the Web application. The application is registered with a URL prefix of */hotdotcom-internal* so the home page can be accessed with the URL *http://host/hotdotcom-internal/index.jsp* as shown in Figure 7–15. If you've forgotten how to assign URL prefixes to Web applications, review Section 4.1 (Registering Web Applications).

Now, the main home page has no security protections and consequently does not absolutely require an entry in *web.xml*. However, many users expect URLs that list a directory but no file to invoke the default file from that directory. So, I put a `welcome-file-list` entry in *web.xml* (see Listing 7.24 in the next section) to ensure that *http://host/hotdotcom-internal/* invokes *index.jsp*.

Listing 7.23 *index.jsp* (Top-level home page)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>hot-dot-com.com!</TITLE>
<LINK REL=STYLESHEET
      HREF="company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">hot-dot-com.com!</TABLE>
<P>
<H3>Welcome to the hot-dot-com intranet</H3>
Please select one of the following:
<UL>
  <LI><A HREF="financial-plan.html">Financial Plan</A>.
      Available to all employees.
  <LI><A HREF="business-plan.html">Business Plan</A>.
      Available only to corporate executives.
  <LI><A HREF="employee-pay.jsp">Employee Compensation Plans</A>.
      Available to all employees.
</UL>
</BODY>
</HTML>
```

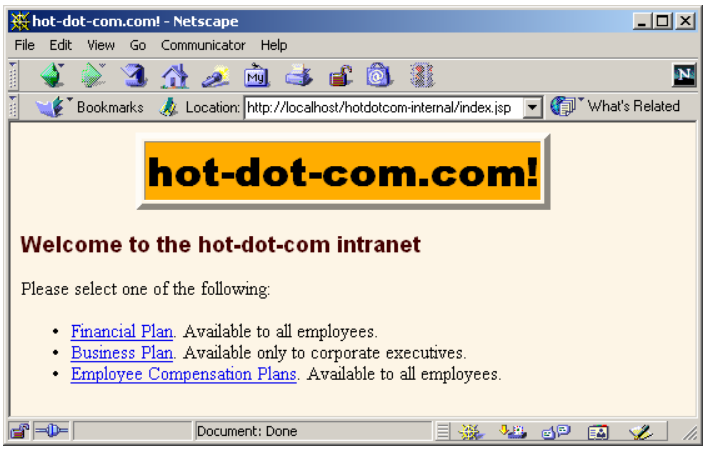

Figure 7–15 Home page for the hot-dot-com.com intranet.

The Deployment Descriptor

Listing 7.24 shows the complete deployment descriptor used with the `hotdot-com-internal` Web application. Again, remember that the order of the sub-elements within the `web-app` element of *web.xml* is not arbitrary—you must use the standard ordering. For details, see Section 5.2 (The Order of Elements within the Deployment Descriptor).

The deployment descriptor specifies several things:

- URLs that give a directory but no filename result in the server first trying to use *index.jsp* and next trying *index.html*. If neither file is available, the result is server specific (e.g., a directory listing).
- URLs that use the default servlet mapping (i.e., *http://host/hotdotcom/servlet/ServletName*) are redirected to the main home page.
- The *financial-plan.html* page can be accessed only by company employees or executives.
- The *business-plan.html* page can be accessed only by company executives.

Listing 7.24 *WEB-INF/web.xml* (Complete version for *hot-dot-com.com* intranet)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <!-- A servlet that redirects users to the home page. -->
  <servlet>
    <servlet-name>Redirector</servlet-name>
    <servlet-class>hotdotcom.RedirectorServlet</servlet-class>
  </servlet>

  <!-- Turn off invoker. Send requests to index.jsp. -->
  <servlet-mapping>
    <servlet-name>Redirector</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- If URL gives a directory but no filename, try index.jsp
       first and index.html second. If neither is found,
       the result is server specific (e.g., a directory
       listing). -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <!-- Protect financial plan. Employees or executives. -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Financial Plan</web-resource-name>
      <url-pattern>/financial-plan.html</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>employee</role-name>
      <role-name>executive</role-name>
    </auth-constraint>
  </security-constraint>
```

Listing 7.24 *WEB-INF/web.xml* (Complete version for *hot-dot-com.com* intranet) *(continued)*

```
  <!-- Protect business plan. Executives only. -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Business Plan</web-resource-name>
      <url-pattern>/business-plan.html</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>executive</role-name>
    </auth-constraint>
  </security-constraint>

  <!-- Tell the server to use BASIC authentication. -->
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Intranet</realm-name>
  </login-config>
</web-app>
```

The Password File

Password files are not specific to Web applications; they are general to the server. Listing 7.25 shows the password file used by Tomcat for this Web application. It defines three new users: `gates` and `ellison` in the `employee` role and `mcnealy` in the `executive` role.

Listing 7.25 *install_dir/conf/tomcat-users.xml* (Three new users)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tomcat-users>
  <user name="john" password="nhoj"
        roles="registered-user" />
  <user name="jane" password="enaj"
        roles="registered-user" />
  <user name="juan" password="nauj"
        roles="administrator" />
  <user name="juana" password="anauj"
        roles="administrator,registered-user" />
  <user name="gates" password="llib"
        roles="employee" />
  <user name="ellison" password="yrral"
        roles="employee" />
  <user name="mcnealy" password="ttocs"
        roles="executive" />
</tomcat-users>
```

The Financial Plan

Listing 7.26 shows the first of the protected pages at the `hotdotcom-internal` site. Figure 7–16 shows the dialog box presented by Netscape to unauthenticated users who attempt to access the page. Figures 7–17 and 7–18 show unsuccessful and successful login attempts, respectively.

Listing 7.26 *financial-plan.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Financial Plan</TITLE>
<LINK REL=STYLESHEET
      HREF="company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Financial Plan</TABLE>
<P>
<H3>Steps:</H3>
<OL>
  <LI>Make lots of money.
  <LI>Increase value of stock options.
  <LI>Make more money.
  <LI>Increase stock option value further.
</OL>
</BODY>
</HTML>
```

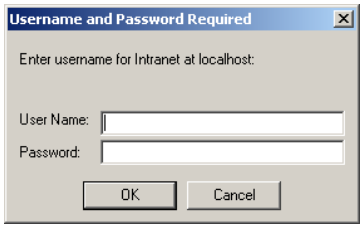

Figure 7–16 Unauthenticated users who attempt to access protected resources are presented with a dialog box.

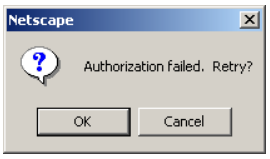

Figure 7–17 A failed login attempt.

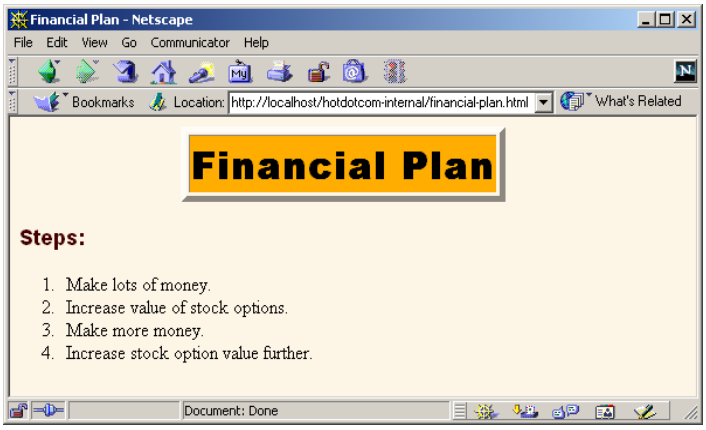

Figure 7–18 A successful login attempt.

The Business Plan

The financial plan of the previous section is available to all employees and executives. The business plan (Listing 7.27), in contrast, is available only to executives. Thus, it is possible for an authenticated user to be denied access to it. Figure 7–19 shows this result. OK, so you have access to more than one username/password combination. You were authenticated as a user with restricted privileges. You now want to log in as a user with additional privileges. How do you do so? Unfortunately, the answer is: quit the browser and restart. Boo. That's one of the downsides of BASIC authentication.

Figure 7–20 shows the result after the browser is restarted and the client logs in as a user in the `executive` role (`mcnealy` in this case).

Listing 7.27 *business-plan.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Business Plan</TITLE>
<LINK REL=STYLESHEET
      HREF="company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Business Plan</TABLE>
<P>
<H3>Steps:</H3>
<OL>
  <LI>Inflate name recognition by buying meaningless ads
      on high-profile TV shows.
  <LI>Decrease employee pay by promising stock options instead.
  <LI>Increase executive pay with lots of perks and bonuses.
  <LI>Get bought out before anyone notices we have no
      business plan.
</OL>
</BODY>
</HTML>
```

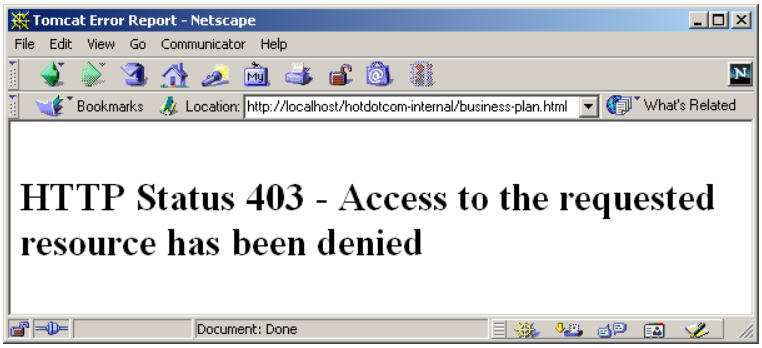

Figure 7–19 Attempt to access the business plan by an authenticated user who is not in the `executive` role. This result is different from that of failed authentication, which is shown in Figure 7–17.

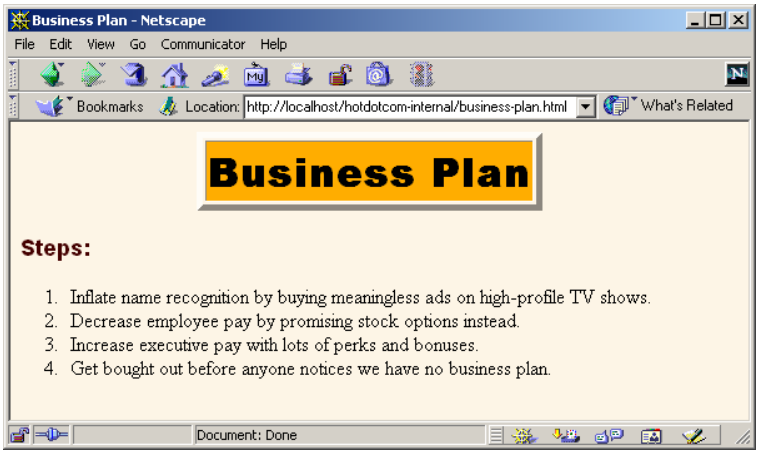

Figure 7–20 Attempt to access the business plan by an authenticated user who is in the `executive` role.

The Redirector Servlet

As it currently stands, the `hotdotcom-internal` application has no protected servlets. So, it is not absolutely necessary to disable the invoker servlet and redirect requests that are sent to *http://host/hotdotcom-internal/servlet/something*. However, it is a good idea to plan ahead and disable the invoker servlet as a matter of course in *all* Web applications that have restricted resources.

This application uses the same redirector servlet (Listing 7.19) and `url-pattern` entry in *web.xml* (Listing 7.24) as does the external `hotdotcom` application.

7.5 Configuring Tomcat to Use SSL

Servlet and JSP containers are not required to support SSL, even in fully J2EE-compliant application servers or with version 2.3 of the servlet specification. Of the three servers that are free for development use and are most commonly discussed throughout the book (Tomcat, JRun, and ServletExec), only Tomcat can directly support SSL. With JRun and ServletExec, traffic between the client and the Web server can be encrypted with SSL, but local communication between the Web server and the JRun or ServletExec plugin does not use SSL. So, Web applications that rely on SSL are not necessarily portable.

Nevertheless, SSL is extremely useful, and many applications make use of it. For example, many application servers are self-contained; they do not have a servlet/JSP

plugin that is separate from the main Web server. In addition, some server plugins use SSL even for the communication between the Web server and the plugin. The BEA WebLogic plugin and IBM WebSphere support this very useful capability, for example.

Even with Tomcat, the default configuration lacks SSL support; you have to install some extra packages to add this support. This section summarizes the steps necessary to do so. For more details, see *http://jakarta.apache.org/tomcat/tomcat-4.0-doc/ssl-howto.html*.

1. **Download the Java Secure Socket Extension (JSSE).** Obtain it from *http://java.sun.com/products/jsse/index-102.html*. Note that this step is unnecessary if you are using JDK 1.4 or later; JSSE is integrated into the JDK in such case.
2. **Put the JSSE JAR files in Tomcat's `CLASSPATH`.** JSSE consists of three JAR files: *jcert.jar*, *jnet.jar*, and *jsse.jar*. The server needs access to all of them. The easiest way to accomplish this step is to put the JAR files into *jdk_install_dir/jre/lib/ext*, thereby making JSSE an installed extension. Note that the Tomcat documentation describes another approach: setting the `JSSE_HOME` environment variable. As of Tomcat 4.0, this approach fails because of a bug in the way Tomcat processes the variable (Tomcat erroneously looks for *jsse.jar* in *JSSE_HOME* instead of *JSSE_HOME/lib*).
3. **Create a self-signed public key certificate.** SSL-based servers use X509 certificates to validate to clients that they are who they claim to be. This prevents attackers from hacking DNS servers to redirect SSL requests to their site. For real-world use, the certificate needs to be signed by a trusted authority like Verisign. For testing purposes, however, a self-signed certificate is sufficient. To generate one that will be valid for two years (730 days), execute the following:

   ```
   keytool -genkey -alias tomcat -keyalg RSA -validity 730
   ```

   The system will prompt you for a variety of information starting with your first and last name. For a server certificate, this should be the server's name, not your name! For example, with a server that will be accessed from multiple machines, respond with the hostname (*www.yourcompany.com*) or the IP address (*207.46.230.220*) when asked "What is your first and last name?" For a development server that will run on your desktop, use *localhost*. Remember that, for deployment purposes, self-signed certificates are not sufficient. You would need to get your certificate signed by a trusted Certificate Authority. You can use certificates from `keytool` for this purpose also; it just requires a lot more work. For testing purposes, however, self-signed certificates are just as good as trusted ones.

Core Approach

Supply the server's hostname or IP address when asked for your first and last name. Use localhost *for a desktop development server.*

The system will also prompt you for your organization, your location, a keystore password, and a key password. Be sure to use the same value for both passwords. The system will then create a file called *.keystore* in your home directory (e.g., */home/username* on Unix or *C:\Documents and Settings\username* on Windows 2000). You can also use the `-keystore` argument to change where this file is created.

For more details on `keytool` (including information on creating trusted certificates that are signed by a standard Certificate Authority), see *http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html*.

4. **Copy the keystore file to the Tomcat installation directory.** Copy the *.keystore* file just created from your home directory to *tomcat_install_dir*.
5. **Uncomment and edit the SSL connector entry in *tomcat_install_dir/conf/server.xml*.** Look for a commented-out `Connector` element that encloses a `Factory` element referring to the `SSLServerSocketFactory` class. Remove the enclosing comment tags (`<!--...-->`). Change the port from 8443 to the default SSL value of 443. Add a `keystoreFile` attribute to `Factory` designating the name of the keystore file. Add a `keystorePass` attribute to `Factory` designating the password. Here is an example (class names shortened and line breaks added for readability).

```
<Connector className="...http.HttpConnector"
           port="443" minProcessors="5"
           maxProcessors="75" enableLookups="true"
           acceptCount="10" debug="0"
           scheme="https" secure="true">
  <Factory className="...net.SSLServerSocketFactory"
           clientAuth="false" protocol="TLS"
           keystoreFile=".keystore"
           keystorePass="your-password"/>
</Connector>
```

6. **Change the main connector entry in *tomcat_install_dir/conf/server.xml* to use port 443 for SSL redirects.** Use the `redirectPort` attribute to specify this. Here is an example.

```
<Connector className="...http.HttpConnector"
           port="80" minProcessors="5" maxProcessors="75"
           enableLookups="true" redirectPort="443"
           acceptCount="10" debug="0"
           connectionTimeout="60000"/>
```

7. **Restart the server.**
8. **Access https://localhost/.** (Note that this URL starts with *https*, not *http*.) With Netscape, you should see initial warnings like those of Figures 7–21 through 7–25. Once you have accepted the certificate, you should see the Tomcat home page (Figure 7–26). With Internet Explorer, you will see an initial warning like that of Figure 7–27. For future requests, you can suppress the warnings by viewing and importing the certificate (Figures 7–28 and 7–29).

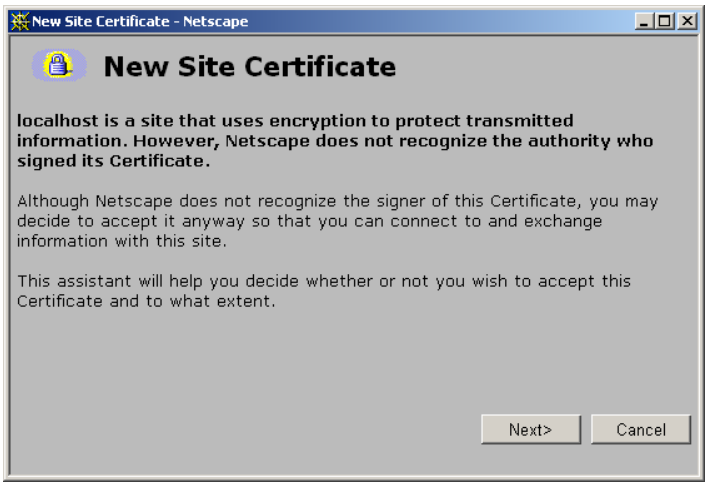

Figure 7–21 First new-certificate window supplied by Netscape.

Figure 7–22 Second new-certificate window supplied by Netscape. Self-signed certificates are for testing purposes only.

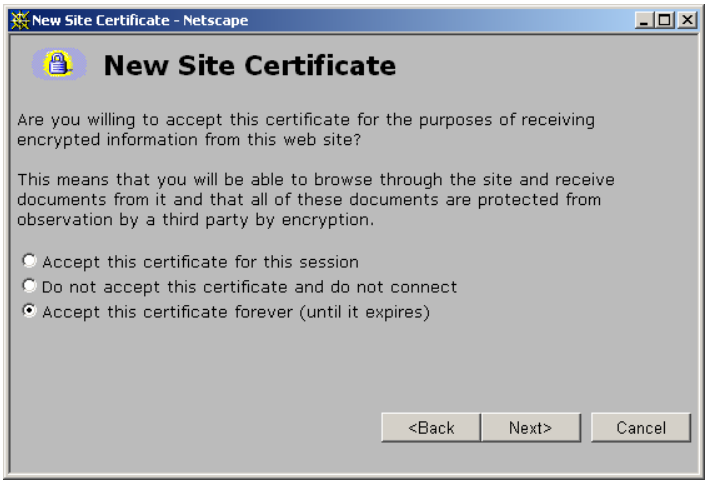

Figure 7–23 Third new-certificate window supplied by Netscape. By choosing to accept the certificate permanently you suppress future warnings.

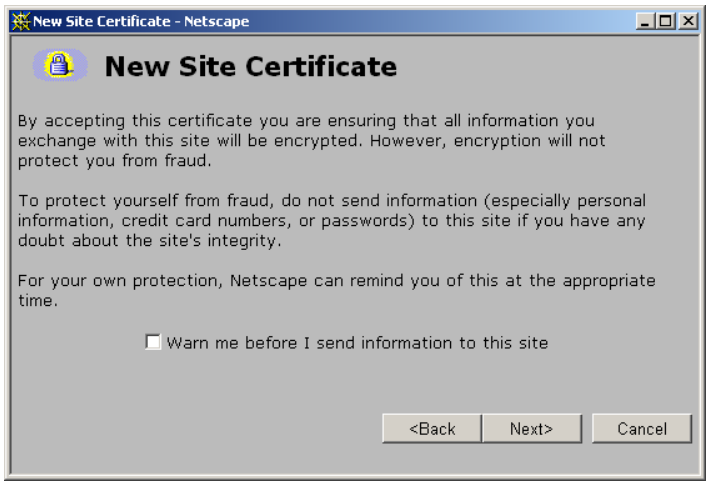

Figure 7–24 Fourth new-certificate window supplied by Netscape.

Figure 7–25 Fifth new-certificate window supplied by Netscape.

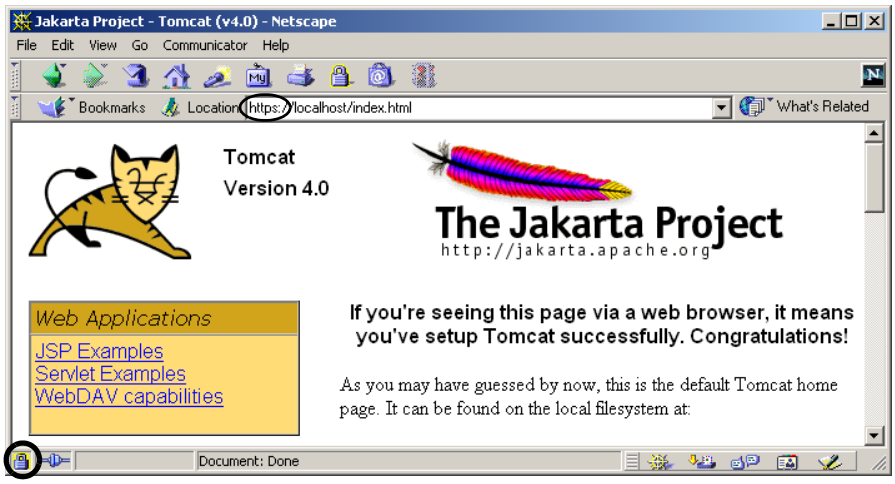

Figure 7–26 A successful attempt to access the Tomcat home page using SSL.

Figure 7–27 New-certificate page for Internet Explorer. View and import the certificate to suppress future warnings. Again, self-signed certificates would not be trusted in real-world applications; they are for testing purposes only.

Figure 7–28 Result of choosing View Certificate in Internet Explorer's new-certificate page.

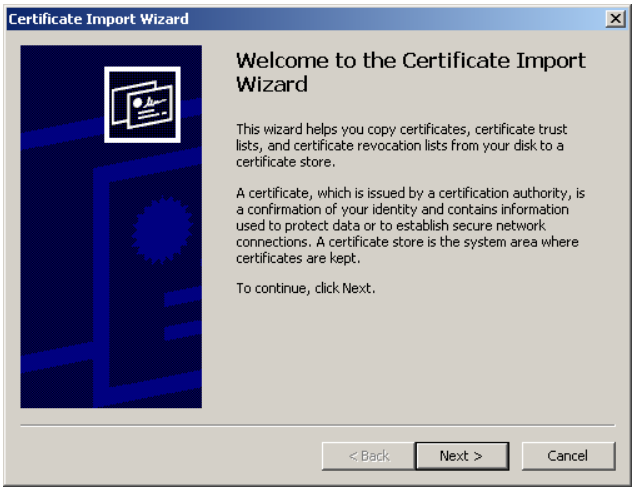

Figure 7–29 Importing self-signed certificates in Internet Explorer lets you suppress warnings that the certificate comes from a company that you have not chosen to trust.